BRANDING FRAMEWORK

TECHNICAL FIELD

[0001] Embodiments of the present invention relate to the field of software architecture. In particular, embodiments of this invention relate to a branding framework for a software product, including an operating system.

BACKGROUND OF THE INVENTION

[0002] As software products become larger and more complex, it is increasingly more difficult to create new products based on existing software products and to service these products as demanded by market and competitive needs. An operating system (OS) is an example of a relatively large, complex software product.

[0003] The operating system manages and schedules the resources of a computer or device in which it resides, and provides various function modules and interfaces that may be used by applications to accomplish various tasks. A conventional computer generally executes its operating system to manage various aspects of the computer as it is running. For example, the operating system is typically responsible for managing access to storage devices as well as input and/or output devices, and controlling the execution of one or more additional applications. Installation usually occurs before the computer executes the operating system (e.g., by copying multiple files from a distribution medium such as a CDROM onto a hard disk of the computer).

[0004] A conventional operating system has a large number of files (e.g., thousands) for instructions and/or data. Such instructions, when executed by the computer, provide the operating system's functionality. Typically, branding information appears throughout many of the operating system's files to provide users with a consistent, professional user experience. For example, the operating system very often presents user interfaces in which a product name, logo, bitmap image, or the like appears. Branding changes are usually made at key phases of product development and continue even after a product such as an operating system is released. These phases include the internal production cycle, beta (external) releases, and the final retail product.

[0005] The process of manually applying consistent brand identification throughout a large, complex software product such as an operating system tends to be complicated, time-consuming, and error-prone. This is due at least in part to the difficult task of finding and replacing every single branded area in the product. The manual branding process is particularly problematic because it must be repeated each time branding changes are made within a product cycle, when different versions of a product are released, when new products are created based on an existing product, and so forth. Branding is desired to be ubiquitous and, thus, conventional processes for making changes, testing the changes, and fixing branding bugs can require many thousands of hours for a large software product.

[0006] As products are developed and updated, it is extremely difficult to know whether all of the possible areas in which branding information appears are displaying properly. Products with visible inconsistencies in branding are highly undesirable.

[0007] Accordingly, a branding framework is desired to address one or more of these and other disadvantages and to allow, among other things, building improved operating systems and other software products.

SUMMARY OF THE INVENTION

[0008] Embodiments of the invention overcome one or more deficiencies in the prior art by simplifying the process by which branding elements are applied to a software product such as an operating system. The invention provides, among other things, a completely new way to handle every aspect of product branding. For instance, the invention centralizes all branding information and, thus, permits branding changes to be applied by making one change in one place rather than making countless changes throughout any given software product. Moreover, aspects of the invention benefit third parties that have purchased licensing rights to customize products with their own branding.

[0009] The invention in at least one of its embodiments encapsulates branding information into a single branding component, which decouples branding from core operating system code. Further aspects of the invention take advantage of a componentized architecture for applying branding elements to a software product. Moreover, the features of the present invention described herein are less laborious and easier to implement than currently available techniques as well as being economically feasible and commercially practical.

[0010] Briefly described, a computerized method embodying aspects of the invention is for use in branding a software product. The method includes assigning a namespace to each of a plurality of resource files. The resource files each contain one or more branding resources. The method also includes grouping the resource files according to the assigned namespaces and executing an interface to call a group of resource files as a function of a selected namespace. The called group of resource files is searched for one or more of the branding resources to be installed in the software product.

[0011] Another embodiment of the invention relates to computer-readable media including a plurality of centrally stored resource files and a branding engine. The resource files each contain one or more branding resources and has a namespace assigned thereto. The resource files are grouped according to the assigned namespaces. The branding engine calls a group of resource files as a function of a selected namespace and searches the called group of resource files for one or more of the branding resources to be installed in the software product.

[0012] Yet another method of branding a software product includes assigning a namespace to each of a plurality of resource files and embedding, in each of the resource files, metadata identifying branding resources contained in the resource files. The method also includes executing an interface to call at least one of the resource files as a function of a selected namespace and searching the called resource file for one or more of the branding resources to be installed in the software product based on the embedded metadata.

[0013] Computer-readable media having computer-executable instructions for performing methods of branding embody further aspects of the invention.

[0014] Alternatively, the invention may comprise various other methods and apparatuses.

[0015] Other features will be in part apparent and in part pointed out hereinafter.

BRIEF DESCRIPTION OF THE DRAWINGS

[0016] FIG. 1 is a block diagram illustrating an exemplary computer according to embodiments of the present invention.

[0017] FIG. 2 is an exemplary block diagram illustrating a component and corresponding manifest of FIG. 1.

[0018] FIG. 3 to FIG. 7 illustrate an exemplary component definition schema according to embodiments of the present invention.

[0019] FIG. 8 is a block diagram of a branding framework embodying aspects of the invention.

[0020] FIG. 9 is an exemplary flow diagram illustrating process flow according to one embodiment of the invention.

[0021] FIG. 10 illustrates an exemplary design process according to embodiments of the present invention for selecting components to be included in a software product.

[0022] FIG. 11 illustrates an exemplary programming object model according to embodiments of the present invention.

[0023] FIG. 12 is a block diagram illustrating exemplary component dependencies for branding a software product.

[0024] FIG. 13 is a block diagram illustrating an exemplary componentization architecture for use with the component definition schema of FIGS. 3-8.

[0025] FIG. 14 is a block diagram illustrating exemplary components of a computer for use FIG. 1.

[0026] Corresponding reference characters indicate corresponding parts throughout the drawings.

DETAILED DESCRIPTION OF THE INVENTION

[0027] Referring now to the drawings, FIG. 1 illustrates an exemplary computing device 150 in accordance with certain embodiments of the invention. The computing device 150 is illustrated as having a software product, such as operating system 152, and one or more application programs 154. The operating system 152 is the fundamental software control program for computing device 150, performing various functions including providing a user interface, managing the execution of one or more applications 154, and controlling the input of data from and output of data to various input/output (I/O) devices.

[0028] In this instance, branding information in the form of a product name, logo, bitmap image, or the like appears throughout many of the files of operating system 152.

Branding changes are usually made at key phases of product development and continue even after a product such as an operating system is released. These phases include the internal production cycle, beta (external) releases, and the final retail product. Embodiments of the invention simplify the process by which branding elements are applied to a software product such as operating system 152. The invention provides, among other things, a completely new way to handle every aspect of product branding. For instance, the invention centralizes all branding information and, thus, permits branding changes to be applied by making one change in one place rather than making countless changes throughout any given software product. Moreover, aspects of the invention benefit third parties that have purchased licensing rights to customize products with their own branding. As will be described in greater detail below, embodiments of the invention encapsulate branding information into a single branding component, which decouples branding from core operating system code. Further aspects of the invention take advantage of a componentized architecture for applying branding elements to a software product.

[0029] In addition to a large number of files, an operating system (e.g., operating system 152) usually has a large number of dependencies among files. For instance, many files may require the installation of one or more other files for their intended functionality to be carried out. Although the operating system's manufacturer may know of such dependencies at the time of installation, it can be difficult for a user, administrator, or other software developer to learn about these dependencies. This can prove troublesome, as software developers may not know what data or instructions in other files affect the particular file(s) with which they are concerned. Similarly, an administrator or user troubleshooting a malfunctioning computer may not know which files are applicable to the problem without knowledge of the dependencies.

[0030] Those skilled in the art are familiar with conventional operating systems, in which both server and client OS products are built from a giant central file that contains all of the binary files. In this example of a generally monolithic conventional operating system, a setup executable must specify which files are included in the different products. Thus, relationships between various parts of each product are difficult to understand. This lack of information hinders the ability to service pieces of an operating system during its lifecycle. In other words, it becomes more difficult for developers and others to keep track of all the different modules and files of the operating system to ensure that they will work properly together and individually as the operating system becomes bigger and more complicated.

[0031] As an example, a particular OS product may be sold as a "home" version while a premium product with additional features may be sold as a "professional" version. In this instance, the home and professional products will often include different branding information and other references interspersed throughout the many binary files making up the respective products. Unfortunately, creation and servicing of products is extremely laborious and requires extensive testing due to relationships between binary files and the existence of these branding references (each of which must be changed for a new product).

[0032] The operating system may be modified (e.g., updated or serviced) in any of a wide variety of manners, such as by adding or replacing one or more particular binary files, by any of a wide variety of people (e.g., a user, administrator, software developer other than the operating system developer, etc.). When such modifications occur, it increases the difficulty of identifying dependencies among files existing on the computer. For example, a change to a single file may necessitate a change to other dependent files. Further, it becomes even more difficult to troubleshoot a malfunctioning computer or update the operating system because the user or administrator cannot easily know exactly what functionality is or should be installed on the computer.

[0033] In the illustrated embodiment of FIG. 1, application programs 154 represent one or more of a wide variety of software application programs that may be executed on computing device 150. Examples of such application programs 154 include educational programs, reference programs, productivity programs (e.g., word processors, spreadsheets, databases), recreational programs, utility programs (e.g., communications programs), etc. Application programs 154 may be installed on computing device 150 by the user, or alternatively pre-installed by the manufacturer and/or distributor of computing device 150.

[0034] The operating system 152 of FIG. 1 separates its functionality into multiple components 156 such as component #1 through component #N in the illustrated embodiment (see also FIG. 2 and FIG. 3). Each component 156 has a corresponding manifest 158 such as manifest #1 through manifest #N, respectively. The components 156 include a collection of one or more files (or file identifiers). The files may include software instructions such as an executable file, a dynamic-link library (DLL), or a component object module (COM). The files may also include data for use by one or more of the components 156. In one implementation, the files (e.g., data and/or instructions) corresponding to particular functionality of the operating system 152 are grouped together in the same component 156. For example, there may be a games component, a communications component, and a file system component. The grouping of files that result in the componentization may be static or alternatively may change over time. In one example, updates to operating system 152 may result in selected files from certain components 156 being removed and added to other components 156.

[0035] Referring further to FIG. 1, each manifest 158 includes information describing the corresponding component 156. Any of a wide variety of metadata regarding the corresponding component 156 may be included in each manifest 158. In one implementation, the manifest 158 identifies the version of the corresponding component 156 as well as which other components 156, if any, the corresponding component 156 is dependent on. By way of example, in order for one or more files in component 156 to properly execute, one or more other files (e.g., a DLL file) from another component may need to be installed on computing device 150. In this example, manifest 158 would indicate that component 156 depends on the other component.

[0036] Due to the complexity and size of existing operating systems, it is becoming increasingly difficult to create new products to meet market or competitive needs. Today's operating systems are more monolithic in nature and, thus, it can be difficult to understand the relationships between various constituent parts. The lack of this information makes servicing an operating system very difficult during the lifecycle of the product. To overcome these and other problems, embodiments of the present invention componentize the operating system. In other words, the invention permits representing a software product as a collection of components. Aspects of the invention involve several abstractions used to build an operating system in a componentized way to facilitate creation of new products and servicing the existing product. The abstractions can be applied to any software product including application programs and any operating system.

[0037] An exemplary component definition schema introduces several abstractions, namely, components (also referred to as assemblies), categories, features, packages, products, and SKUs (stock keeping units). In this instance, component 156 represents a reusable, sharable, self-describing atomic unit of distribution, servicing, and/or binding. It is the most basic abstraction that describes the component itself and all relevant information (i.e., metadata) used for installing, servicing, and/or binding to necessary resources in a declarative manner.

[0038] As described in greater detail below with respect to an embodiment of the invention, a category object 162 (see FIG. 4) represents a mechanism to group a common set of components 156; a feature object 164 (see FIG. 5) represents a composition of components 156 and/or features 164 and is used as a building block for creating a product object 166 (see FIG. 7); and a package object 168 (see FIG. 6) represents a composition of components 156, features 164, categories 162, and/or other packages 168. As an example, the package object 168 is used to group a set of components 156 for administrative purposes. Further, the product object 166 in this embodiment represents a top level composition of features 164 and/or components 156 and a SKU object 170 (see FIG. 7) represents a shipping mechanism for products 166 (e.g., one SKU 170 may contain multiple products 166).

[0039] Referring now to FIG. 8, a client binary component 174 according to the embodiments of the invention utilizes a branding framework for installing, updating, modifying, and/or servicing branding resources applied to the software product. The branding framework consolidates branding into a single location, so all products that include branding information will benefit. In this instance, the branding binaries are divided into two parts, namely, a branding engine 176 and a plurality of branding resource files 178. The branding engine 176 accesses branding resources contained in the resource files 178. In other words, the branding framework of the invention utilizes branding engine 176 to apply branding changes throughout a software product. The branding framework also supports the creation of custom branding components (i.e., resources), to be used as needed for products that contain unique brand elements in addition to, for example, a more general brand such as the OS brand. For example, each branding resource file is a Win32® resource only DLL or other application programming interface (API) in which one or more branding resources reside. As described above, branding resources include strings and images representative of product names, logos, bitmap images, and the like.

[0040] The branding resource files 178 are separated into namespaces, which in turn are grouped into components. In one embodiment of the invention, the binaries of branding engine 176 contain an API for calling a group of resource files 178 (i.e., component) as a function of a selected namespace. Based on the specified namespace, branding engine 176 searches a DLL for requested branding resources. Advantageously, all of the branding resource files 178 need not be installed. Rather, this embodiment of the invention installs only the branding components that are required by the installed components 156.

[0041] Post-release, branding changes may also be required for each different SKU 170 that is released (e.g., premium version, home version, etc.). Over the long term, the branding framework of the invention enables all SKUs 170 to be serviced with the same binary files when the brand does not need to be changed. Service packs, QFE releases, and the like are understood to be brand agnostic as well.

[0042] Referring further to the binary containing the API interface, branding engine 176 knows in which DLL to look for a requested resource using the namespace specified. The following are examples of unmanaged APIs of branding engine 176:

[0043]
```
INT BrandLoadString(LPCTSTR NameSpace, // Name space of the
branding (sub) component that contains the requested resource
          INT Id,
          LPTSTR buffer,
          INT size);
```
Same behavior as the Win32 SDK API LoadString.

[0044]
```
HANDLE BrandLoadImage(LPCTSTR NameSpace, // Name space of the
branding (sub) component that contains the requested resource
          LPCTSTR lpszName,     // image to load
          UINT uType,           // image type, for now only IMAGE_BITMAP
                                will be supported
          int cxDesired,        // desired width
          int  int cyDesired,   // desired height
```

int UINT _fuLoad_ // load options

);

Same behavior as the Win32 SDK API LoadImage. The caller has to destroy the returned object (same as with LoadImage).

[0045] HBITMAP BrandingLoadBitmap(LPCTSTR NameSpace,

LPCTSTR lpszName)

Same behavior as the Win32 SDK API LoadBitmap. The caller has to destroy the returned object (same as with LoadBitmap).

[0046] HCURSOR WINAPI BrandingLoadCursor(LPCTSTR NameSpace,

LPCTSTR lpszName)

Same behavior as the Win32 SDK API LoadCursor. The caller has to destroy the returned object (same as with LoadCursor).

[0047] HICON WINAPI BrandingLoadIcon(LPCTSTR NameSpace,

LPCTSTR lpszName)

Same behavior as the Win32 SDK API LoadIcon. The caller has to destroy the returned object (same as with LoadIcon).

[0048] The branding engine 176 uses a NameSpace parameter in this embodiment to find the branding component (i.e., resource DLL 178). For an example, see the branding resource file below.

[0049] A "helper API,"

LPTSTR BrandingFormatString(LPCTSTR inString)

represents an exemplary branding resource file 178. The helper API is used for messages containing product names. The string that the caller passes in may contain replaceable parameters for the product names. The following are exemplary product name parameters:

[0050] %WINDOWS_GENERIC% for the generic product name (IDS_WINDOWS_GENERIC);

%WINDOWS_SHORT% for the short product name (IDS_WINDOWS_SHORT, e.g.: Windows XP® operating system);

%WINDOWS_LONG% for the long product name (IDS_WINDWOS_LONG, e.g.: Windows XP ® Home Edition operating system);

%WINDOWS_PRODUCT% for the SKU name (IDS_WINDOWS_PRODUCT, e.g., Professional);

%WINDOWS_COPYRIGHT% for the copyright string (IDS_WINDOWS_COPYRIGHT, e.g., Copyright © 1983-2003 Microsoft Corporation);

%MICROSOFT_COMPANYNAME% for the Microsoft company name (IDS_MICROSOFT_COMPANY, e.g., Microsoft);

%WINDOWS_VERSION% for the version year (IDS_WINDOWS_VERSION, e.g., Version 2003).

[0051] The function in this example replaces the parameters with the corresponding product string, and returns the new string. A GlobalAlloc function, for example, is performed on the string. The caller frees the string using GlobalFree. The input string is not touched. Any other replaceable parameter (e.g. %ld (for sprintf) or %1!ld! (for FormatMessage)) stays "as is." The caller can call his/her preferred function to replace those before or after calling into BrandingFormatString. Alternatively, the caller can call BrandingLoadString to get the product strings and then pass them to his/her preferred function. BrandingFormatString is only a helper API that is provided for the convenience of the users of branding engine 176.

[0052] The following is an example of an input string: "%WINDOWS_LONG% is the best product ever." The output would be: "Windows XP® Home Edition is the best product ever" if the computer is running the Windows XP® Home Edition operating system.

[0053] Referring now to FIG. 9, one aspect of the invention involves customized branding by a third party such as an original equipment manufacturer (OEM). Branding engine 176 supports OEM custom branding, which is accomplished by an extensible markup language (XML) file contained in the branding resource DLL 178. The XML file, also referred to as a branding manifest, describes the resources contained in an associated branding resource DLL 178. When a resource is added to the branding resource DLL (i.e., resource file 178), the XML data for the DLL is updated for the new resource(s).

[0054] In one embodiment, one of the data fields in the XML file indicates whether a particular branding resource to be installed in the software product can be provided/overwritten by an OEM or other third party. As an example, the data contains the following information: ResourceType, ResourceID, and Overwrite. ResourceType describes identifies the branding resource by type (e.g., String = 6, Bitmap = 2) whereas ResourceID provides an identifier. Overwrite, which has a default value of NO, indicates whether a third party is permitted to provide a resource for the identified branding information. In addition, the manifest can also include embedded metadata describing characteristics other characteristics of the resource (e.g., size, length, color, format, etc.).

[0055] Beginning at 182, FIG. 9 describes the process of obtaining third party resources. A binary or other file(s) (e.g., component 156) first requests a branding resource. Advantageously, maintaining branding rules in the resources themselves prevents resource corruption. The component calls into a branding API of branding engine 176 at 184. In this example, the branding API, which may be different for different types of resources, is modeled after a Win32® unmanaged API. It is to be understood that the interface could also be managed (e.g., such as with the .NET framework).

[0056] At 186, branding engine 176 checks for a branding manifest based on, for example, namespace and resource ID. The specified namespace maps to a specific DLL in this example. At this point, branding engine 176 attempts to locate the DLL in its centralized location and to determine if the ID for the requested resource is known to the DLL. Branding engine 176 further examines the manifest for an override instruction at 188. With override set to a NO value, the regular resource is returned at 190. On the other hand, if override is set to a YES value, a third party will be granted permission to change an aspect of the visual appearance of the software product.

[0057] Proceeding to 192, branding engine 176 checks for the presence of third party override DLL. As before, branding engine 176 searches a known location based on the specified namespace for the override DLL. If the override DLL does not exist or does not specify that it wants to override the regular resource, operations return to 190. If the override DLL is available, the third party resource is returned at 194.

[0058] The following is an example of the data file (i.e., .the branding manifest):

[0059]
```
<Branding>
    <Resource ResourceType="2" ResourceID="100" Overwrite="No"/>
    <Resource ResourceType="2" ResourceID="101" Overwrite="No"/>
    <Resource ResourceType="2" ResourceID="110" Overwrite="No"/>
    <Resource ResourceType="2" ResourceID="111" Overwrite="No"/>
    <Resource ResourceType="6" ResourceID="10" Overwrite="Yes"/>
    <Resource ResourceType="6" ResourceID="11" Overwrite="Yes"/>
    <Resource ResourceType="6" ResourceID="12" Overwrite="Yes"/>
    <Resource ResourceType="6" ResourceID="13" Overwrite="Yes"/>
</Branding>
```

[0060] If a resource is not provided in the branding resource DLL in the first instance, an entry in the above data file is required. However, the resource may be provided by a third party rather than in the branding resource DLL. For example, an entry in the data file is used to indicate that the OEM/third party can overwrite a particular resource such as an OEM logo on the system property page of Windows XP® operating system. In this example, the branding resource DLL need not have a default if the OEM chooses not to provide the branding resource.

[0061] The component definition described herein may also be used to describe application components generally rather than operating system components specifically. In other words, any application may be described using this component definition.

Accordingly, the branding framework may be applied to any operating system, application program, or other software product.

[0062] The componentization architecture described herein defines the concepts, component repository, and programming model for managing components both during design-time and run-time. There are several strategic benefits of componentization including agility, speed, supportability, and increased revenue opportunities. Creating a new software product, represented by a product or SKU object, is made much easier, even across client and server lines. It is a relatively simple task of selecting the right components and providing some additional configuration to build a new product. The agility in creating new products in a relatively short time provides the ability to compete efficiently and avoid missing a market opportunity. For example, it is possible to offer additional components to later add to a current product, which allows a customer to upgrade to a premium product, which increases the overall number of licenses, etc. All of these additional activities may result in additional revenues for the products.

[0063] Componentization also facilitates reducing the number of product images that an original equipment manufacturer (OEM) or corporation must maintain, which can lead to great cost savings. According to at least one embodiment of the invention, most of the component information is declarative such that components can be installed in an offline manner. This reduces the time it takes to install the whole product on a target device and results in great cost reductions for OEMs, corporations, end users and the like as well as increased customer satisfaction.

[0064] Those skilled in the art recognize the importance of service as part of the product cycle. Servicing a componentized product according to embodiments of the invention is relatively simple because it is easy to assess the impact of the changes based on the declarative information provided by each component. It is also much easier to test the components in a more isolated environment to improve the testing efficiency. In turn, this reduces the overall fragility in the software product.

[0065] The component definition schema described herein covers the information that component 156 describes in order to install, upgrade service, and bind to appropriate resources. In this regard, FIG. 2 illustrates exemplary component 156 and corresponding manifest 158 in accordance with certain embodiments of the invention. According to embodiments of the invention, each component 156 is represented by a corresponding manifest 158. Component 156 includes at least one file 174 and may optionally include more than one (*n*) files. Although illustrated as files in FIG. 2, component 156 may alternatively include pointers or other identifiers of one or more of files 174 rather than the actual files.

[0066] The component 156 corresponds to manifest 158. In the illustrated example, manifest 158 includes a component identifier that identifies component 156 (e.g., by name or some other unique identifier). This correspondence can alternatively be maintained in different manners, such as inclusion of an identifier (not shown) of manifest 158 in component 156, storage of both component 156 and manifest 158 (or identifiers thereof) in a data structure that maintains an inherent correspondence between component 156 and manifest 158, etc. Manifest 158 may be an extensible markup language (XML) document.

[0067] As shown in the exemplary listing of FIG. 2, manifest 158 also includes a dependent component list that identifies zero or more components that component 156 is dependent on. The identified dependent components are those components that also need to be installed as part of the operating system image in order for component 156 to function properly. In the illustrated example, the identifiers identify components that are necessary for component 156 to function properly, but alternatively may include components that should be included (i.e., components preferred by component 156 to have as part of the operating system but which are not necessary). In addition to identity and dependencies, manifest 156 in this example also describes a number of other details of component 156, namely, general information (including owner, tester, developer, description, etc.), files, registry Information, settings (configuration), memberships, and other information.

[0068] The manifest 158 in an alternative embodiment may also include a priority order and a version indicator to aid in installation and/or upgrading.

[0069] Alternatively, some or all of the information maintained in manifest 158 may be maintained in different locations. By way of example, some or all of the information may be incorporated into component 156 of FIG. 1.

[0070] The example of APPENDIX A further illustrates aspects of the invention with respect to manifest 158.

[0071] The following description provides further details regarding the major abstractions used in a componentization architecture exemplifying aspects of the invention.

[0072] As described above, the object referred to as component 156 (or assembly) represents a reusable or sharable self-describing atomic unit of distribution, servicing, and binding. In the embodiment of FIG. 3, components 156 may depend on other components 156 and/or features 164 to run, i.e., they exhibit dependencies relative to these other objects. Components 156 may also be members of different open groups, namely, categories 162. In this instance, a developer-friendly, extensible, multi-part property "bag" referred to as "identity" identifies each component 156. The component identity has the following attributes, for example: name, version, processor architecture, language (e.g., "us-eng" or "jpn"), build type (e.g., free or debug), and originator's identification. The name attribute takes the form of a locale independent string that describes the particular component 156 in one embodiment. A four-part version number, which generally follows a "major.minor.build.revision" format, for example, is provided by the version attribute of the identity. Cryptographically secure information that allows the component identity to be secure is found in the originator's ID attribute of the identity.

[0073] As set forth above, each component 156 is made up of one or more files as well as an associated manifest 158. Manifest 158 describes the details of component 156, as shown in FIG. 3. In the embodiment of FIG. 3, the object referred to as component or assembly may have dependencies with respect to another component 156, a category 162, and/or a feature 164. In particular, FIG. 3 shows that component 156 may be dependent on one or more of the binary files found in zero or more categories 162, zero or more features 164, and/or zero or more other components 156. In addition, component 156 may declare membership in category 162.

[0074] The category object 162 shown in FIG. 4 defines an open group in which membership information is present with the member of the category rather than being present in a category manifest. For example, one category 162 may be established to group one or more components 156 related to text editors such as Notepad and Wordpad. Categories 162 are also identified using an identity mechanism similar to that of components 156. In one embodiment, categories 162 have details such as identity, general information, and other information.

[0075] In the exemplary relationship diagram of FIG. 4, category 162 may be used to group zero or more features 164, zero or more components 156, and/or zero or more packages 168. As described above, the category object 162 represents an open group into which other objects declare their membership.

[0076] Referring now to FIG. 5, the feature object 164 represents a composition of components 156 and/or other features 164. Features 164 are used as software product building blocks and can be added or removed from installation. Rather than depending upon other objects, features 164 have inclusive relationships with respect to components 156, other features 164, and categories 162. In other words, features 164 do not have dependencies according to this embodiment. As an example, one feature object 164 represents a consolidated Web browser feature and another feature object 164 represents a media player application, both of which are shipped with an operating system. Feature 164 consists of identity, general information (e.g., owner, tester, developer, description, etc.), memberships, compositions, and other information. In one embodiment of the invention, the feature identity is similar to the component identity in structure and is used to identify each feature 164. Features 164 may be members of multiple different categories 162 (see FIG. 4).

[0077] FIG. 5 further illustrates that feature 164 includes zero or more other features 164 and/or components 156 and that feature 164 may declare membership in zero or more of the categories 162.

[0078] FIG. 6 diagrammatically illustrates the package object 168. As shown, package 168 is a composition of components 156, features 164, categories 162, and other packages 168. In contrast to categories 162, packages 168 are closed groups. Packages 168 are primarily used for administrative purposes. For example, a home version of an OS may be shipped with four different packages 168 indicated by “cab1”, “cab2”, “cab3” and “cab4”. The package objects 168 in this example are groups formed for setup purposes. Packages 168 specify dependencies only on other packages 168in the embodiment of FIG. 6. Each package 168 according to the componentization definitions described herein consists of details such as identity, general information, membership, compositions, dependencies, and other information.

[0079] Referring further to FIG. 6, package 168 represents a grouping of zero or more other packages 168, categories 162, features 164, and/or components 156. In this instance, package 168 may belong to an open group such as category 162 or depend from a closed group such as another package 168.

[0080] The product object 166 of FIG. 7 represents a top level composition of features 164 and/or components 156. It also has properties that are used to configure the composed features 164 and components 156. According to embodiments of the invention, a product designer selects one or more features 164 for product 166 (see FIG. 13). The product object 166 contains details on identity, general information, compositions, and other information. SKU object 170 represents the shipping medium for products 166 (i.e., what products are included on a disc for shipping). Suitable shipping media include floppies or compact discs and web download cabinet files. Also, SKU 170 may consist of more than one product 164 (see FIG. 13). At least one embodiment of the invention generates a SKU manifest using a SKU designer tool. Similarly to the other abstractions described above, SKU 170 contains identity, general information, products, and other information.

[0081] FIG. 10 provides an example of grouping various components 156 of OS binary files to define features 164. The features 164 are then grouped to define products 166, which are in turn included for shipping with SKU 170. Different computers may have different operating system images that are based on the same operating system. For example, different OEMs may customize the same operating system in different ways (e.g., so that the operating system boots with an initial screen identifying the OEM, different default settings may be used, etc.). Different functionality may also be included (e.g., screen savers, backgrounds or themes, software applications such as communications programs, games, etc.). This additional functionality can be provided by the OEM, or alternatively other manufacturers or distributors. Thus, many different components may be part of the operating system of which only a subset are actually installed on a particular computer as an operating system image. An OEM-specific component and manifest set, on the other hand, includes additional components that can be installed by the OEM as part of the operating system image.

[0082] The operating system can be updated for any of a wide variety of reasons. By way of example, bug fixes to certain files of certain components may be available, new functionality (e.g., replacement or additional files) in a component may be available, new components may be available, etc.

[0083] Additionally, a new component may be installed as part of the operating system along side a previous component rather than replacing it. This allows different applications to use whichever version of the component they prefer (or are programmed to use).

[0084] By way of example, an OEM may offer various basic computer configurations corresponding to home use, business use, server use, and so forth. Each one of the configurations in this example is based on the same operating system but includes different functionality. Additional networking and communications functionality may be included in the server configuration that is not included in either the home or business configurations, and additional games or audio playback functionality may be included in the home configuration that is not included in the business or server configurations. FIG. 10 shows an exemplary premium product that includes a basic home product.

[0085] FIG. 11 shows first level abstractions of the programming model according to embodiments of the invention. In particular, the exemplary diagram illustrates different first level objects (or classes), using, for example, Unified Modeling Language (UML). The objects are exposed to programmers using UML in this example. Each of the interfaces directly maps to the first level concepts of the componentization.

[0086] The operating system installation process is simplified greatly by the componentization of the operating system. OEM-specific functionality can be easily added to a computer by including the appropriate component and corresponding manifest. Further, updates to the operating system for improved functionality, bug fixes, and the like can be easily incorporated into the installation process by simply replacing the corresponding components. Alternatively, an additional update component and manifest set may be available to an operating system installation station that includes such updates.

[0087] Referring now to further aspects of the branding framework of the invention, internal branding scenarios are generally based on manual processes in at least one embodiment. As an example, components 156 may be authored to include product branding. This includes components 156 that display product branding in their UI as well as components 156 that have no unique branding UI in addition to the existing generic product branding elements. In this instance, the component author enables components 156 to display branding at points during the componentization process, such as planning and implementation.

[0088] At the planning phase, for example, the author of component 156 may check its UI for branding elements and find several instances of generic product branding elements. If the author does not find any branding that is unique to the particular component, he or she may conclude that component 156 can use the existing generic product branding elements in the branding category 162. In this instance, the author notes that component 156 will need to express a dependency on the branding engine 178 APIs.

[0089] At the implementation phase, for example, the author finds the branding resources inside the branding category 162, which lists all of the resource files 178 divided by branding resource component (DLLs). In the alternative, a root of all branding binaries may be available to the author. During implementation, the author chooses the branding resource components from the branding category 162 and codes the UI for component 156. At this point, the author adds a dependency from component 156 to the branding API. At build time, the specified branding elements are displayed in the UI.

[0090] In yet another embodiment, component authors may require unique or custom branding elements instead of generic branding (e.g., for a premium product). Referring now to FIG. 12, the branding framework embodying aspects of the invention permits component authors to specify dependencies to branding engine 176 for creating a software product having custom branding elements.

[0091] FIG. 13 illustrates exemplary componentization architecture for use with the invention. The extensible architecture shown in FIG. 13 permits the building of an OS installation, including installing and uninstalling components, and enables product/SKU agility. Moreover, the architecture provides infrastructure for servicing a running OS and an OS image, provides an inventory of OS components; and lives on the installed system. The componentization architecture provides a generalized framework for creation and management of components. It also allows for building a run-time image from a configuration in a generic manner. A component management interface (CMI) is the programming model interface that may be used by different consumers to access functionality exposed by the componentization architecture. The tools can be generally divided into the following classes: design-time tools; run-time tools; and build-time tools. Design-time tools are the tools used at the design time (e.g., a component designer to design or create components and target designer to design or create a configuration). Run-time tools are used to manage the existing components on a run-time image (installation). This might include functionality such as an option component manager (OCM), which manages the components installed on a system. Another example of a run-time tool can be a servicing client that talks with a remote server to get component updates and updates the components installed on an installation. Build-time tools are used in the build lab. They are primarily used to create a run-time image from a configuration.

[0092] As shown in FIG. 13, CMI has a core layer and a utility layer. The CMI core layer consists of different blocks of functionality that are generally exposed and consumed as well known interfaces. This layer consists of a core CMI object model, a repository and file repository, and serializer/de-serializer. The core CMI object model block exposes multiple different first class abstractions as interfaces to CMI clients for ease of programming. They generally reflect componentization abstractions in a relatively easy to use format. For example, IAssembly abstracts the "component" concept. The abstractions in this embodiment do not have any functionality associated with them. They are a collection of attributes that can be retrieved and set through the methods exposed on the abstractions. The following list includes some of the abstractions that are exposed through an interface in the core CMI object model: IAssembly; IFeature; ICategory; IPackage; IProduct; ISku; IConfiguration; IRepository.

[0093] A component repository in the example of FIG. 13 is used for managing components that are present in one of the repositories. As described above, each component has metadata (i.e., information about the data) and the actual data (i.e., files of a component). Metadata is accessible through a metadata repository interface

(IRepository) whereas data is accessible through a file repository interface (IFileRepository). In one embodiment, the metadata repository interface is implemented using a relational database to store and index the metadata of components. The file repository interface allows clients to get to the component data (files) in a consistent way.

[0094] The component metadata and data interfaces are split under the component repository interface in FIG. 13 to allow for different pluggable metadata repository implementations based on different scenarios. For example, on a design machine one could use a SQL metadata repository since available storage and memory might not be constrained but on a run-time system metadata repository might be backed by a small footprint database (e.g., registry) to take care of tight memory requirements.

[0095] A serializer and deserializer interface allows first class objects in CMI to be serialized to and deserialized from a text file. For example, an XML serializer and deserializer reads and writes XML files. The serializer and deserializer generally read and write out carriers and configurations as described below.

[0096] The files that carry any of the first class abstractions in serialized format are referred to as either carriers or manifests. The carriers provide a means of populating the component repository data, i.e., tools create or edit a serialized instance of component in a carrier file and the component repository interface allows for the import of the carrier into the component repository.

[0097] The primary advantage of using a carrier for information exchange is that it allows the tools to be decoupled from the component repository. Another advantage is that while importing the carrier information, the data can be imported into the component repository in a more consistent (or complete) format. The serializer and deserializer interface segregation also allows for other kinds of carriers (e.g., INF) to be implemented.

[0098] Configuration is a serialized representation of the CMI's IConfiguration object, which represents a collection of components and settings used to build a run-time image. The reason configuration is serialized in a separate file rather than a carrier file is that configuration cannot be imported into the database.

[0099] The CMI utility layer consists of blocks of functionality that frequently changes. The blocks of functionality are exposed and consumed as well known interfaces in the embodiment of FIG. 13. The utility layer includes blocks for an installer, an upgrader, a dependency resolver, the CMI object model, and clients. The installer block has the logic for installing (or building) and removing a particular componentization abstraction exposed by the core layer. For example, IAssembly is installed and removed from the OS installation in an online or offline manner. The logic of installing and removing assemblies is present in this block. All the core abstractions are installed and removed using "IInstallable" interface. The implementation of these methods changes for each core abstraction. For example, for assembly there will be an installer abstraction called "IAssemblyInstaller". IAssemblyInstaller aggregates "IAssembly" interface and implements "IInstallable" to install and uninstall an assembly instance.

[0100] This type of particular functionality binding with core abstractions allows the implementation of assembly install logic to change as needed without affecting the core IAssembly interface. The same holds true for other core abstractions also.

[0101] The upgrader block has the logic for upgrading and downgrading a particular core abstraction on a windows installation. For example, IAssembly is upgraded or downgraded from the OS installation in an online or offline manner. The logic of upgrading and downgrading assemblies is present in this block. All the core abstractions are upgraded and downgraded using "IUpgradable" interface.

[0102] There is a separate implementation of these methods for each core abstraction. Since install and uninstall functionality is needed during the upgrade process, "IUpgradable" inherits from "IInstallable" to re-use the existing functionality of install and uninstall. For example, the assembly has an upgrader abstraction called "IAssemblyUpgrader". IAssemblyUpgrader aggregates "IAssembly", "IAssemblyInstaller" and implements "IUpgradable" to upgrade and downgrade an assembly instance on a windows installation. Again, this kind of binding allows the implementation of assembly upgrade logic to change as needed without affecting the core IAssembly interface and install or uninstall logic. The same holds true for upgrade functionality of other core abstractions also.

[0103] The dependency resolver block implements the dependency resolution logic for a particular core abstraction. In this instance, for each core abstraction this block has logic in terms of dependency resolution either in an automated or manual fashion. The dependency resolver is extensible for the client programs to extend the default dependency resolution functionality as needed in a particular client context. This block's functionality is exposed through "IDependencyResolver" interface. The method of resolving dependencies returns a tree of CMI objects or instance objects based on the scenario where it is being used. Dependency resolution is generally done with respect to a configuration or a repository according to at least one embodiment of the invention. The repository can be either installable or installed repository.

[0104] Generally, on run-time systems, dependency resolutions happen against an installed repository and existing configuration, whereas in a design-time scenario the dependency resolution happens against an installable repository and a current configuration which is being edited.

[0105] The CMI object model is an aggregated form of CMI core object model and different pieces of functionality exposed in the utility layer. The core CMI object model is also exposed out for authoring tools to manipulate the abstractions directly while serializing and de-serializing the carriers (or manifests).

[0106] One of the key things to notice in the above architecture is that the same programming model is advantageously exposed for design-time, run-time and build-time scenarios. This helps in keeping the object model consistent with respect to different requirements and helps in programmer's productivity. This also helps in re-using a single implementation of CMI to be used for different scenarios such as design and run-time and is therefore more maintainable in comparison to different implementation for design-time and run-time scenarios.

[0107] FIG. 14 shows one example of a general purpose computing device in the form of a computer 70. In one embodiment of the invention, a computer such as the computer 70 is suitable for use as computer 150.

[0108] In the illustrated embodiments, computer 70 has one or more processors or processing units 72 and a system memory 74. In the illustrated embodiment, a system bus 76 couples various system components including the system memory 74 to the processors 72. The bus 76 represents one or more of any of several types of bus structures, including a memory bus or memory controller, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus also known as Mezzanine bus.

[0109] The computer 70 typically has at least some form of computer readable media. Computer readable media, which include both volatile and nonvolatile media, removable and non-removable media, may be any available medium that can be accessed by computer 70. By way of example and not limitation, computer readable media comprise computer storage media and communication media. Computer storage media include volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. For example, computer storage media include RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium that can be used to store the desired information and that can accessed by computer 70. Communication media typically embody computer readable instructions, data structures, program modules, or other data in a modulated data signal such as a carrier wave or other transport mechanism and include any information delivery media. Those skilled in the art are familiar with the modulated data signal, which has one or more of its characteristics set or changed in such a manner as to encode information in the signal. Wired media, such as a wired network or direct-wired connection, and wireless media, such as acoustic, RF, infrared, and other wireless media, are examples of communication media. Combinations of the any of the above are also included within the scope of computer readable media.

[0110] The system memory 74 includes computer storage media in the form of removable and/or non-removable, volatile and/or nonvolatile memory. In the illustrated embodiment, system memory 74 includes read only memory (ROM) 78 and random access memory (RAM) 80. A basic input/output system 82 (BIOS), containing the basic routines that help to transfer information between elements within computer 70, such as during startup, is typically stored in ROM 78. The RAM 80 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 72. By way of example, and not limitation, FIG. 16 illustrates operating system 84, application programs 86, other program modules 88, and program data 90.

[0111] The computer 70 may also include other removable/non-removable, volatile/nonvolatile computer storage media. For example, FIG. 14 illustrates a hard disk drive 94 that reads from or writes to non-removable, nonvolatile magnetic media. FIG. 14 also shows a magnetic disk drive 96 that reads from or writes to a removable, nonvolatile magnetic disk 98, and an optical disk drive 100 that reads from or writes to a removable, nonvolatile optical disk 102 such as a CD-ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 84, and magnetic disk drive 96 and optical disk drive 100 are typically connected to the system bus 76 by a non-volatile memory interface, such as interface 106.

[0112] The drives or other mass storage devices and their associated computer storage media discussed above and illustrated in FIG. 14, provide storage of computer readable instructions, data structures, program modules and other data for the computer 70. In FIG. 14, for example, hard disk drive 94 is illustrated as storing operating system 110, application programs 112, other program modules 114, and program data 116. Note that these components can either be the same as or different from operating system 84, application programs 86, other program modules 88, and program data 90. Operating system 110, application programs 112, other program modules 114, and program data 116 are given different numbers here to illustrate that, at a minimum, they are different copies.

[0113] A user may enter commands and information into computer 70 through input devices or user interface selection devices such as a keyboard 120 and a pointing device 122 (e.g., a mouse, trackball, pen, or touch pad). Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are connected to processing unit 72 through a user input interface 124 that is coupled to system bus 76, but may be connected by other interface and bus structures, such as a parallel port, game port, or a universal serial bus (USB). A monitor 128 or other type of display device is also connected to system bus 76 via an interface, such as a video interface 130. In addition to the monitor 128, computers often include other peripheral output devices (not shown) such as a printer and speakers, which may be connected through an output peripheral interface (not shown).

[0114] The computer 70 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 134. The remote computer 134 may be a personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to computer 70. The logical connections depicted in FIG. 16 include a local area network (LAN) 136 and a wide area network (WAN) 138, but may also include other networks. LAN 136 and/or WAN 138 can be a wired network, a wireless network, a combination thereof, and so on. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets, and global computer networks (e.g., the Internet).

[0115] When used in a local area networking environment, computer 70 is connected to the LAN 136 through a network interface or adapter 140. When used in a wide area networking environment, computer 70 typically includes a modem 142 or other means for establishing communications over the WAN 138, such as the Internet. The modem 142, which may be internal or external, is connected to system bus 76 via the user input interface 134, or other appropriate mechanism. In a networked environment, program modules depicted relative to computer 70, or portions thereof, may be stored in a remote memory storage device (not shown). By way of example, and not limitation, FIG. 14 illustrates remote application programs 144 as residing on the memory device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

[0116] Generally, the data processors of computer 70 are programmed by means of instructions stored at different times in the various computer-readable storage media of the computer. Programs and operating systems are typically distributed, for example, on floppy disks or CD-ROMs. From there, they are installed or loaded into the secondary memory of a computer. At execution, they are loaded at least partially into the computer's primary electronic memory. The invention described herein includes these and other various types of computer-readable storage media when such media contain instructions or programs for implementing the steps described herein in conjunction with a microprocessor or other data processor. The invention also includes the computer itself when programmed according to the methods and techniques described herein.

[0117] For purposes of illustration, programs and other executable program components, such as the operating system, are illustrated herein as discrete blocks. It is recognized, however, that such programs and components reside at various times in different storage components of the computer, and are executed by the data processor(s) of the computer.

[0118] Although described in connection with an exemplary computing system environment, including computer 70, the invention is operational with numerous other general purpose or special purpose computing system environments or configurations. The computing system environment is not intended to suggest any limitation as to the scope of use or functionality of the invention. Moreover, the computing system environment should not be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary operating environment. Examples of well known computing systems, environments, and/or configurations that may be suitable for use with the invention include, but are not limited to, personal computers, server computers, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable consumer electronics including mobile telephones, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

[0119] Embodiments of the invention may be described in the general context of computer-executable instructions, such as program modules, executed by one or more computers or other devices. Generally, program modules include, but are not limited to, routines, programs, objects, components, and data structures that perform particular tasks or implement particular abstract data types. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote computer storage media including memory storage devices.

[0120] In operation, computer 70 executes computer-executable instructions such as those described herein to assign a namespace to each of the resource files 178 and to group them according to the assigned namespaces. Computer 70further executes an interface to call a group of resource files as a function of a selected namespace and searches the called group for one or more of the branding resources to be installed in the software product. In another embodiment, computer 70 executes computer-executable instructions such as those described herein for embedding, in each of the resource files 178, metadata identifying branding resources contained in the resource files. In this instance, computer 70 searches a called resource file for one or more of the branding resources to be installed in the software product based on the embedded metadata.

[0121] Those skilled in the art will note that the order of execution or performance of the methods illustrated and described herein is not essential, unless otherwise specified. That is, it is contemplated by the inventors that elements of the methods may be performed in any order, unless otherwise specified, and that the methods may include more or less elements than those disclosed herein.

[0122] Information in this document, including uniform resource locator and other Internet web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred.

[0123] When introducing elements of the present invention or the embodiments thereof, the articles “a,” “an,” “the,” and “said” are intended to mean that there are one or more of the elements. The terms “comprising,” “including,” and “having” are intended to be inclusive and mean that there may be additional elements other than the listed elements.

[0124] In view of the above, it will be seen that the several objects of the invention are achieved and other advantageous results attained.

[0125] As various changes could be made in the above constructions and methods without departing from the scope of the invention, it is intended that all matter contained in the above description and shown in the accompanying drawings shall be interpreted as illustrative and not in a limiting sense.

APPENDIX A

[0126] Data Storage and Format: Authoring tools create the component, feature, category, package, product and SKU. In this example, they are represented in an XML file (called a carrier or manifest). Each carrier contains only one instance of features or categories or packages or products or SKUs. An example of a notepad manifest follows:

```
<?xml version="1.0" encoding="UTF-16"?>
<!-- edited with XMLSPY v5 U (http://www.xmlspy.com) by vijayj (ms) -->
<!-- edited with XML Spy v4.4 U (http://www.xmlspy.com) by Vijay Jayaseelan (ms) -->
<assembly manifestVersion="1.0" authors="vijayj" company="Microsoft"
copyright="Microsoft" displayName="Notepad" lastUpdateTimeStamp="2002-07-
31T09:23:00" owners="none" released="false" testers="none"
supportInformation="http://www.microsoft.com" description="Unicode and non-unicode
text file editor." xmlns="urn:schemas-microsoft.com:asm.v2">
        <assemblyIdentity name="notepad" version="1.0.0.0"
processorArchitecture="x86" language="neutral" buildType="release"/>
        <dependency>
                <dependentCategory name="Notepad Language Category"
version="1.0.0.0" processorArchitecture="x86" language="*" buildType="release"
selection="one"/>
        </dependency>
        <dependency>
                <dependentAssembly>
                        <assemblyIdentity name="Windows Shell" version="1.0.0.0"
processorArchitecture="x86" language="*" buildType="release"/>
                </dependentAssembly>
        </dependency>
        <file name="notepad.exe" sourcePath="%_NTTREE%\"/>
        <memberships>
                <categoryMembership name="Text Editors" version="1.0.0.0"
processorArchitecture="x86" language="*" buildType="release"/>
        </memberships>
        <registryKeys>
                <registryKey keyName="HKCU\Notepad\Settings\"/>
                <registryKey keyName="HKCU\Notepad\Settings\Font\">
                        <registryValue name="Name" valueType="REG_SZ"
value="Arial"/>
                        <registryValue name="Size" valueType="REG_DWORD"
value="10"/>
                </registryKey>
        </registryKeys>
        <propertyTypes>
                <propertyType name="DefaultFont" valueType="string" access="public"
readOnly="false" value="Arial" regularExpression="(Arial)|(Lucida Console)|(Courier)">
```

```
                        <registryLocation keyName="HKCU\Notepad\Settings\Font\">
                                <registryValue name="Name" valueType="REG_SZ"
value="Arial"/>
                        </registryLocation>
                </propertyType>
        </propertyTypes>
</assembly>
```